Fuzzing the Latest NTFS in Linux with Papora: An Empirical Study

Edward Lo*‡, Ningyu He†‡, Yuejie Shi*, Jiajia Xu*, Chiachih Wu*, Ding Li†, and Yao Guo†✉

*Amber Group
†Key Lab on HCST (MOE), School of Computer Science, Peking University
‡Co-first authors

***Abstract*—Recently, the first feature-rich NTFS implementation, NTFS3, has been upstreamed to Linux. Although ensuring the security of NTFS3 is essential for the future of Linux, it remains unclear, however, whether the most recent version of NTFS for Linux contains 0-day vulnerabilities. To this end, we implemented Papora, the first effective fuzzer for NTFS3. We have identified and reported 3 CVE-assigned 0-day vulnerabilities and 9 severe bugs in NTFS3. Furthermore, we have investigated the underlying causes as well as types of these vulnerabilities and bugs. We have conducted an empirical study on the identified bugs while the results of our study have offered practical insights regarding the security of NTFS3 in Linux.**

***Index Terms*—fuzzing, file system, NTFS**

I. Introduction

NTFS [1] was developed by Microsoft as the native file system for Windows NT. Decades later, along with the rapid growth of the market share of Windows, numerous hard disks are formatted as NTFS, whose fully read-write support should be taken into consideration for other operating systems, e.g., Linux kernel. NTFS3, as the first feature-rich implementation of the impactful NTFS file system, landed in Linux in late 2021. Albeit the potential benefit, integrating a new component, especially a file system, is extremely likely to introduce bugs or even vulnerabilities. Unfortunately, to the best of our knowledge, there is no systematic study on the found bugs introduced by the latest NTFS3. Even worse, we find there are even no available tools to discover these bugs. Thus, regarding integrated NTFS3 in the Linux kernel, it is necessary to implement a tool to detect bugs, and conduct a systematic evaluation on them to raise the awareness of the community, especially security researchers.

To close this gap, in this paper, we build the first effective fuzzer, named Papora, for NTFS3. Then, we conduct the first fuzzing-based systematic study on identified bugs. In this whole process, we have to underline that it is particularly challenging in engineering. Although there are several fuzzers for file systems, such as Janus [2] and Hydra [3], they cannot be directly applied to fuzz NTFS3 due to two issues. First, they lack a specific parser for NTFS images to extract metadata, correct checksums and assemble corpus. Moreover, directly adopting existing parsers for NTFS is not feasible because they only validate the integrity of the given image, which is insufficient for fuzzy testing. Lacking such a parser will significantly decrease the performance of a fuzzer due to the burden on I/O issues [4]–[6]. Second, they do not support KASAN [7] on our targeting Linux kernel, making the vulnerability hunting less effective.

Fortunately, Papora has addressed the above two tough nuts to some extent. On the one hand, since the implementation of NTFS is not open-sourced, it is particularly tough to build a feasible image parser for it. Though there is a so-called official documentation, it still lacks lots of technical details. To this end, we manually compare multiple third-party releases and their corresponding documentations, and cross-reference which implementation is consistent with the expected behavior. On the other hand, because fuzzing an image via virtual instances may suffer bug reproduction issues [2], we decided to apply LKL [8], a user space application that can emulate behaviors of the Linux kernel, to load NTFS images. However, LKL is not maintained at all, and KASAN is not integrated inside. Therefore, we firstly ported LKL to the latest version, and revisited the instrumented memory subsystem of LKL and enabled KASAN support with intensive effort. For example, LKL adopts a special architecture, i.e., running with a linear memory layout, which is different from other *memory management unit based* (MMU-based) architectures having KASAN support. We have to manually refactor the KASAN codebase and make it compatible with the no-MMU LKL.

The results have proven that Papora is an effective fuzzer. In total, we have discovered 3 CVE-assigned 0-day vulnerabilities and 9 severe bugs in the latest Linux kernel. We have reported these 12 vulnerabilities/bugs to the maintainers with patches, which have all been confirmed. Moreover, 9 out of them have been merged into the upstream. Our study shows that the latest version of NTFS in Linux still suffers from the problems of out-of-bounds read bugs and null pointer dereference bugs. Moreover, to ring the alarm for the community of the security issues resulting from enabling NTFS support, we have conducted sophisticated and meaningful case studies on representative identified bugs. Based on the case study, we also propose some best practices for security researchers and Linux developers to avoid such bugs. We urge the developers to have a deeper investigation into these two types of bugs and improve their security awareness with our empirical study.

This work is partly supported by the National Key Research and Development Program (No. 2022YFB4501802), the National Natural Science Foundation of China (No. 62172009, No. 62141208), and Amber Group.

2023 IEEE Security and Privacy Workshops (SPW) | 979-8-3503-1236-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/SPW59333.2023.00034

DOI 10.1109/SPW59333.2023.00034

We summarize our contribution as follows:

1) To the best of our knowledge, we have implemented the first fuzzer specifically for NTFS3 in Linux. It is able to effectively and efficiently discover new bugs and vulnerabilities.
2) We have identified 3 CVE-assigned 0-day vulnerabilities and 9 severe bugs in NTFS3, among which 9 were confirmed and fixed on the upstream.
3) We have made an empirical fuzzing-based case study on bugs of NTFS3 in Linux.
4) We have released Papora including the LKL, which is ported to the latest version of the Linux kernel and integrated with KASAN, at https://github.com/ambergroup-labs/papora.

This paper is organized as follows: §II introduces some basic knowledge of NTFS and fuzzing testing. In §III, we detail the challenges for fuzzing an NTFS image, and how we build Papora. Moreover, in §IV, we illustrate the 12 bugs/vulnerabilities we identified, and conduct case studies on representative ones. Based on the results, we have summarized some best practices for users, developers and security researchers in §V. Finally, §VI and §VII illustrate published related work and a discussion of some interesting issues of this paper, respectively.

II. Background

A. NTFS File System

A file system is one of the essential components of an operating system that manages files, folders, links, and the data to efficiently process the read/write requests from users to those items. In 1993, the proprietary New Technology File System (NTFS) [1] developed by Microsoft debuted with the first release of Windows NT. Within the last two decades, various NTFS drivers including the legacy driver [9] and FUSE-backed drivers, Captive [10] and NTFS-3G [11], had been contributed to the Linux kernel for providing alternative ways to access Windows hard drivers on Linux. In late 2021, the presence of NTFS3 [12] developed by Paragon Software [13] finally unleashes the power of NTFS in Linux 5.15 kernel.

1) NTFS Features: As the successor of the File Allocation Table (FAT) file system, except for supporting some advanced features like large volumes, NTFS outperforms FAT in many aspects, especially the *reliability* and *security*. Specifically, the reliability of NTFS file systems can be reflected from two aspects. On the one hand, similar to other journaling file systems, NTFS uses the logging and checkpoint mechanisms to guarantee the consistency of the file system for dealing with unexpected system crashes. On the other hand, a recovery technique, named *cluster remapping*, can also improve the reliability. When a bad sector, located in a cluster, is detected in a read operation, NTFS remaps the cluster to a newly allocated one, and marks the bad one that will no longer be used. As for the security issue, NTFS allows granting access to files or directories in users and groups granularity. Moreover, the Encrypting File System (EFS) allows users to encrypt files on NTFS volumes. In this way, even a bad actor can physically access the hard drive, he cannot decrypt any files in an EFS-enabled NTFS volume without the owner's private key [14].

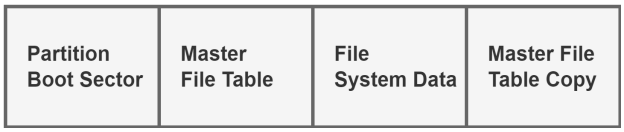

Fig. 1. The Layout of an NTFS File System.

2) NTFS Physical Structure: Fig. 1 illustrates the layout of an NTFS image. The **Partition Boot Sector** (PBS) holds important information for bootstrapping the system. In particular, the boot sector starts with an x86 *jump* instruction which skips some non-executable metadata, e.g., *OEM ID*. And the PBS is also responsible for modifying the program counter to the *Bootstrap Code*. **Master File Table** (MFT) holds the metadata of all files and directories, even including the metadata itself. To ensure the integrity of MFT, NTFS maintains a **Master File Table Copy**, which maintains exact identical data to the MFT. The MFT is composed of multiple entries for NTFS metadata, each of which has fixed functionalities and follows strict syntax. For example, the third entry of the MFT, named *$LogFile*, contains all necessary transactions for a faster recovery when the system crashes. Moreover, the seventh entry, dubbed as *$Bitmap*, maintains a bitmap for all free and unused clusters. Most importantly, MFT also stores some meta-information necessary to retrieve files, like the attributes of a file. In NTFS, each file is stored in clusters that are composed of one or multiple sectors, and structured in a list of attributes, e.g., file name, timestamp, even the file data. The file data that is not contained in the MFT will be stored in **File System Data** [15].

B. Fuzzing

Fuzzing or *fuzzy testing* is an automatic software testing paradigm that tests the target with inputs which are mutated based on the target state and testing results. A naive practice is randomly generating inputs to fuzz the target until it crashes. It might work when the input space is limited but many popular fuzzers guide the input mutation based on code coverage. Coverage-based fuzzers such as AFL [4] and libFuzzer [16] instrument the target in compile time and feed the target states back to the input mutator, which leads the fuzzer to keep exploring new execution paths in the target program.

While fuzzing a user space program, the target takes the mutated inputs from the command line or configuration files and executes in a loop until it crashes. But, it is a different story to fuzz an operating system component like a file system. Specifically, the input space becomes two dimensions, i.e., an image with a file system, and a series of system calls. Traditional kernel fuzzers such as Trinity [17] and Syzkaller [18] generate a series of system calls with parameters as inputs for the target operating system. In particular, Trinity uses the annotation to generate *better-than-random* parameters for each system call to trigger unexpected behaviors more easily.

Syzkaller uses KCOV to collect code coverage and SyzLang to provide context for guided fuzzing. However, the file system image is a more complicated input that cannot be efficiently generated by kernel fuzzers.

File system fuzzing efforts such as Janus [19] and Hydra [20] deal with the problem by extracting metadata from large images with file system-specific parsers. Janus also addresses the aging OS problem with the library OS LKL [8], which helps the fuzzer to quickly reload a clean-slate OS and get rid of irreproducible bugs.

III. Experimental Setup

In this section, we will explain how we implement an NTFS fuzzer and how to efficiently fuzz an NTFS file image.

A. Challenges

As we mentioned in §II-B, fuzzing a file system is challenging, which can be summarized as follows:

- **C1: Disk Image.** For a mainstream fuzzer, e.g., AFL [4], its recommended size of targets is less than 1KB. An empty file system, however, which is embedded in a disk image, often contains more than dozens of megabytes. Directly fuzzing a disk image, which is 1,000x larger than fuzzer's maximum preferred size, will dramatically downgrade the efficiency due to the heavy I/O brought by mutating or booting the given image.
- **C2: Context-aware File Operations.** Except for directly mutating the disk image, file operation is another orthogonal valuable seed. In other words, a series of file operations may also lead to system crashes. Moreover, file operations are context-aware workloads for images. For example, `open()` will actually create a new file on the target image, based on which the following file operations can be performed. Such context-aware file operations not only exponentially increase the exploration space for seed mutation, but also require the corresponding updates on states (e.g., entries in MFT) of the image.
- **C3: Reproduction.** Traditional fuzzers aiming at operating systems often take virtual instances as targets. However, frequently modifying and rebooting virtual instances, or reverting to specific snapshots are extremely time-consuming. Moreover, they may reuse file systems, leading to undetermined and unpredictable states, i.e., aging problems, for file systems, which seriously hinders the reproduction of found bugs.

Janus and Hydra have addressed the above challenges to some extent on multiple file systems, e.g., ext4 and HFS+, except for NTFS. Compared with those targets, the biggest obstacle of fuzzing an NTFS image is the absence of its implementation. Except for Microsoft's documentation where it qualitatively describes the structure and implementation of NTFS, all other releases as we mentioned in §I are third-party implementations. To this end, efficiently and correctly fuzzing an NTFS image is challenging.

B. Overview

According to challenges we introduced in §III-A, Papora is specifically designed to tackle these problems. The overall workflow of Papora is shown in Fig. 2.

As we can see, firstly, an NTFS parser scans the whole given image, and builds a corpus that will be sent to Papora. Our fuzzer tries to mutate both the metadata of the given image and the program consisting of file operations, and updates the status field accordingly (Step 2 & 3). Then, the NTFS parser assembles the updated corpus as an intact mutated image (Step 4), which will be mounted by Linux Kernel Library (LKL) and executed according to the program (Step 5). Finally, the corresponding result of the current round will be outputted (Step 6), and the feedback information will be sent back to Papora to guide the following mutations (Step 7). Technical details are illustrated from the following §III-C to §III-G.

C. Corpus Building

As we can see from §III-B, a corpus is composed of three parts: extracted metadata of the given image, a program consisting of a series of file operations, and a status file. Specifically, a specifically-designed *NTFS parser* (see §III-D) will extract the metadata, i.e., entries in PBS and MFT, out of the image, and condense them into a bulk of data. In this way, the meaningless part for finding new bugs, i.e., File System Data (see Fig. 1), contributing more than 99% of space to the image, will not be included into the corpus. As for the second part, the initial program is an empty sequence of file operations. Last, as we mentioned in §II-A2, attributes of files and directories are stored in the MFT. During scanning the image, these attributes will be packed and maintained in the third part, i.e., the status file, of the corpus. Papora will take the assembled corpus as input, mutate either the metadata or the file operations, mount the image, and execute the program to see if any bugs are triggered. If it is not, the corresponding field of the input corpus will be updated (like the status field should be updated due to file creations), and the corpus will be sent to Papora for the next round fuzzing.

D. NTFS Parser

As we mentioned in **C1**, directly fuzzing an image will face an extreme efficiency problem. Therefore, like previous work [2], [3], we develop a *parser* specifically targeting NTFS to tackle this problem. The responsibility of the parser can be divided into three-folds.

First, the parser can extract all metadata and compress them into a dense bulk of data. For a file system, an image crash after mounting is only due to buggy metadata, accounting for less than 1% space of the image. This indicates that mutating the other 99% space (mainly composed of files' content) is meaningless. As for an NTFS image, metadata is mainly composed of fields in *Partition Boot Sector (PBS)* and *Master File Table (MFT)* (see §II-A2). Therefore, condensing metadata in PBS and MFT where the mutation plays on will not only increase the efficiency for both mutation and the

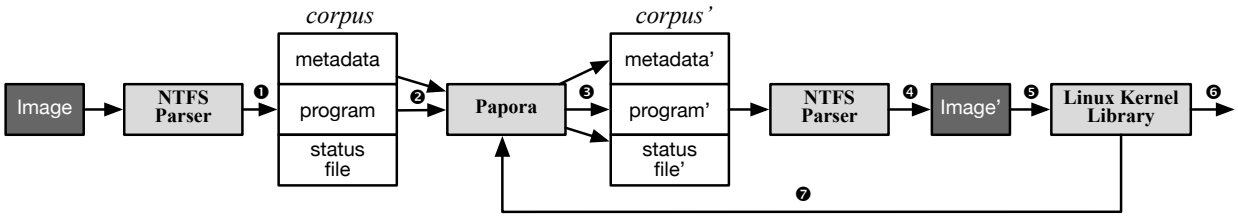

Fig. 2. Overall workflow of Papora.

following fuzzing, but also increase the possibility of finding corrupted metadata related new bugs.

Second, the parser will automatically fix checksums after mutating metadata. For file systems, including NTFS, they all adopt checksums to ensure the integrity and usability of the metadata. A mismatch between checksum literals and in-time calculated checksums will raise an error, resulting in the image cannot be loaded properly. Therefore, after mutating metadata, the parser will re-calculate all the corresponding checksums to guarantee the mutated NTFS file system can pass the static verification on checksums. From §II-A2, we can see that the PBS and MFT are NTFS-specific structures. Papora has some specific checksum fixups on these two data structures. Specifically, the PBS holds important information for boot (see §II-A2). For example, its second field, i.e., OEM ID, is fixed as "NTFS" followed by 4 space characters. Though this field is mutated by Papora accidentally, the parser will recover its original value to make sure mutation has no negative effects on booting. Additionally, MFT stores metadata of all files and directories. The header of each file record contains a *Update Sequence Number* (USN) and a buffer. NTFS requires the last two bytes of each sector of records are copied into the buffer and the USN is written in their place. After booting, NTFS will compare the USN from the header with the last two bytes of each sector. To this end, if Papora mutates headers in MFT, it will modify the corresponding fields to pass sanity checks.

Third, the fuzzing process still performs on an intact image, thus the parser should also be responsible for mapping the mutated metadata in corpus back to the image. To achieve such a goal, during the extracting, the parser will maintain a bitmap in which the offsets of each piece of metadata are kept. Based on the bitmap, the mutated metadata can be filled back into the original slots.

Note that, instead of directly adopting traditional and well-maintained NTFS parsers, like NTFS-3G, implementing our own parser does not mean reinventing a wheel. Although traditional parsers can parse and load NTFS images, they only validate if the given image is valid, or the image will not be loaded successfully. However, as we mentioned above, our parser is responsible for correcting checksums and assembling corpus for the following process. In other words, our parser conducts an extra fixup process based on verifying the validity. Moreover, NTFS-3G is too heavy and inefficient for fuzzing analysis. In summary, it is necessary to implement our own parser in implementing Papora.

E. Fuzzing Image

Papora applies several strategies, e.g., bit/byte flip, and arithmetic operations, on the metadata part of a corpus to mutate it. The strategies can be summarized as follows:

- Flip a bit at random offset, or set an interesting byte/-word/dword value (like min/max valid number, 0, ± 1, and power of 2) at random offset in random endian.
- Randomly add/subtract a random value at random byte/-word/dword offset.
- Overwrite bytes by a random chunk or a random byte for random length at random offset, or by user specified tokens if provided.

After mutating the metadata, the NTFS parser recalculates necessary checksums.

F. Fuzzing File Operations

Except for mutating metadata, Papora also mutates the second part of corpus, i.e., the program consisting of a series of file operations. Papora mutates the program in two strategies: *mutation* and *generation*.

Mutation. Papora prefers this strategy. It randomly picks one file operation in the seed program, then replaces some of its arguments with heuristic values instead of random ones. As we stated in **C2**, these file operations are context-aware. Thus, the selected values should be meaningful for the current image. For example, if the mutated file operation is `fsync()`, which takes a file descriptor as its argument to synchronize its in-core state with the storage device. Papora will pick one of the *opened* file descriptors of proper type. Such a heuristic and context-aware strategy also applies for mutating system calls related to path and extend attributes.

Generation. If Papora cannot increase coverage through mutating the program, it will try to append new file operations to the program with proper arguments. Moreover, the potential side effects of each file operation are taken into consideration, and the program context will be updated accordingly. For instance, `link()` and `mkdir()` may create a new file and directory, while `unlink()` and `rmdir()` have the opposite effects. The program context will record changes introduced by these system calls in the status file of corpus.

G. Linux Kernel Library (LKL)

C3 has stated that if Papora adopts virtual instances to mount the image and execute the program, it will face efficiency and reproduction problems. To this end, Papora builds

its target program, i.e., the executor, with *Linux Kernel Library* (LKL) [8], as a user space program. LKL provides a way for emulating the Linux kernel by compiling the kernel into an object file that can be directly linked by applications. To discover potential vulnerabilities in the latest version, we have upgraded LKL to v6.0[1]. Moreover, to enable detection of illegal memory accesses, we also integrated the *KASAN* [7] provided by [3] into the LKL with several necessary fixups.

Porting the LKL to the latest Linux kernel and integrating KASAN into it are challenging. On the one hand, some interfaces have been introduced, changed or even deprecated. For example, the interface of `copy_thread` is changed, leading to a rewriting of the corresponding function to guarantee the logic correctness. Moreover, header files rearrangement often happens in the upstream. Thus, subsystem maintainers may choose to move some structures or macro definitions to new headers, which may lead to merge conflicts or build errors while porting LKL to the latest kernel. On the other hand, integrating KASAN into LKL needs lots of effort. This is because LKL can be regarded as an architecture with no-MMU (memory management unit) support. In other words, LKL only supports linear memory address, which is conflicted with KASAN initialization flow. Therefore, we have to manually review the flow, adjust or comment out related codes or structures, while maintaining the functionalities of KASAN.

After resolving problems of porting LKL to the latest version, it can bring in several advantages over mounting images through a virtual machine. First, user-space applications are much lighter than the emulator in terms of rebooting. Restarting an application only introduces negligible time compared to resetting a VM instance. Second, VM-based fuzzers may choose to keep running their target programs until the *aging* kernel crashes or hangs. To this end, the initial status of the image is undetermined, which results in irreproducible bugs even with full kernel dump. For security researchers, it is also difficult to obtain the root cause in such indeterministic situations. Papora, however, can restart its executor for every corpus with little overhead, providing a stable and determined kernel state. Third, such a LKL assisted method requires much less computing resources, so it is easy to scale up the fuzzing process by deploying more instances.

IV. Experiment Results

In this section, we will first list all bugs identified by Papora. Then, we will delve deeper and conduct case studies to illustrate the reason behind system crashes. The results show that some severe vulnerabilities may even be used in privilege escalation.

A. Results

We run Papora on a VMware virtual machine with an 8-core CPU and 16GB memory running Ubuntu 16.04. The experiment is conducted intermittently for about 3 months. As the results listed in Table I, Papora has successfully discovered 9 severe bugs and 3 CVE-assigned vulnerabilities (highlighted rows) in the NTFS3 implementation[2]. All identified bugs as well as the corresponding patches have been reported to maintainers, and 9 of them have been merged into the upstream. Additionally, we have categorized these identified bugs into two types. The *Type I* refers to the situation that once the NTFS image is mounted, the system crashes. While the ones under *Type II* can only be triggered by invoking the corresponding system calls after mounting the image.

TABLE I
Identified bugs and vulnerabilities (highlighted rows) by Papora, where NPD and OOB refer to null pointer dereferences and out-of-bound, respectively.

Commit	Bug Type	Root Cause	Upstreamed
Type I			
0b66046	NPD	Sanity check miss	✓
e19c627	OOB Read	Arithmetic overflow	✓
6db6208	OOB Read	Sanity check miss	✓
2681631	NPD	Sanity check miss	✓
c1ca8ef	NPD	Implementation flaw	✓
4f1dc7d (**CVE-2022-48424**)	Heap Corruption	Sanity check miss	✓
bfcdbae	OOB Read	Sanity check miss	✓
e6ffad3	OOB Read	Sanity check miss	
467333a (**CVE-2022-48425**)	Heap Corruption	Type confusion	
f64633f	OOB Read	Sanity check miss	
Type II			
4d42ecd	OOB Read	Sanity check miss	✓
54e4570 (**CVE-2022-48423**)	OOB Write	Sanity check miss	✓

1) Categorized by Bug Type: Over 60% of the bugs identified by Papora are out-of-bounds read bugs. Those bugs are the most dangerous species which could be exploited for leaking kernel information or even corrupting kernel memory. For example, an out-of-bounds read could be used to export the addresses of critical kernel data structures to be corrupted. By exploiting an out-of-bounds write, a process with the mounting capability could escalate its privileges by corrupting function pointers, hijacking the control flow (e.g., jumping to the shellcode or JOP gadgets), and eventually changing credentials data in the `task_struct`.

Around 25% of the bugs identified by Papora are null pointer dereference (NPD) bugs. Those bugs directly crash the target system and make the target hang or reboot depending on the system configuration. By exploiting an NPD, a bad actor could launch denial-of-services attacks on target systems with the mounting capability or the auto-mounting feature enabled.

Except for OOB access and NPD bugs, Papora identified 2 heap corruption bugs which could be developed into use-after-free (UAF) exploits. Specifically, when a memory chunk is allocated in the Linux kernel, a reference pointer is returned by the slab system and all upcoming access would go through that pointer. A typical exploit is filling another victim memory chunk containing function pointers into the intentionally

[1]At the time of writing, v6.0 is the latest version for Linux. Moreover, the LKL project was inactive and the supported kernel version stayed at v5.3.

[2]For simplicity, all these 12 identified issues will be referred by *bugs* if they are mentioned as a whole.

released spot and corrupting the victim memory chunk through the old reference pointer.

2) Categorized by Root Cause: While analyzing those bugs identified by Papora, most of them are due to missed sanity checks on user-controllable data. Specifically, any data field retrieved from a file system image is a chunk of user-controllable data which should always be strictly checked as it would be used as the input of the NTFS3 implementation. For example, a crafted `offset` field could simply lead to an out-of-bounds read if it is not bounded by the size of the allocated memory to cache the metadata. Furthermore, if the `offset` is derived from another number-of-entries field, an overflowed `offset` could be crafted when *number-of-entries* $\times$ *size-of-entry* is large enough. That overflowed `offset` would bypass the sanity check for the bounding `offset` itself. As a result, all arithmetic operations related to user-controllable data should be carefully validated as well.

Papora also identified bugs caused by type confusion [21]. We believe that it is a common type of bug in Linux file system implementations due to the design of inode. In particular, each inode could be interpreted in various ways depending on the states or flags. As shown in Listing 1, the `union` in `struct ntfs_inode` makes each `ntfs_inode` represent either a `dir` or a `file`. Commit `467333a` [22] demonstrates a type confusion case in which the NTFS3 implementation wrongly interprets an `MFT_REC_MFT` file as a directory and corrupts the heap by `kfree()`-ing an invalid pointer.

```
union {
  struct ntfs_index dir;
  struct {
    struct rw_semaphore run_lock;
    struct runs_tree run;
#ifdef CONFIG_NTFS3_LZX_XPRESS
    struct page *offs_page;
#endif
  } file;
};
```

Listing 1. A code snippet of struct `ntfs_inode`

The root cause of `c1ca8ef` bug identified by Papora could be categorized in the *Always-Incorrect Control Flow Implementation* class [23]. In other words, instead of preparing a malicious input, a bad actor could trigger the crash with a normal test case which is missed due to incomplete test coverage. That is exactly the problem we need a customized fuzzer like Papora to cope with.

B. Case Study on Type I

Type I bugs occur during mounting an NTFS disk, whose traces are shown in Fig. 3. As we can see, once invoking the `mount`, the Linux system will trap into the kernel space. Most of the mounting processes are handled by Linux's VFS layer. Because files in Linux are arranged in a tree-like hierarchical structure, the `vfs_get_tree` will call the specialized `ntfs_fs_get_tree` to get its mountable root. Within the implementation of NTFS, the function `ntfs_fill_super` plays a vital role. Specifically, it parses the partition boot sector (see §II-A2) and reads parametric data, e.g., cluster size and maximum size of normal files. It also loads all metadata files

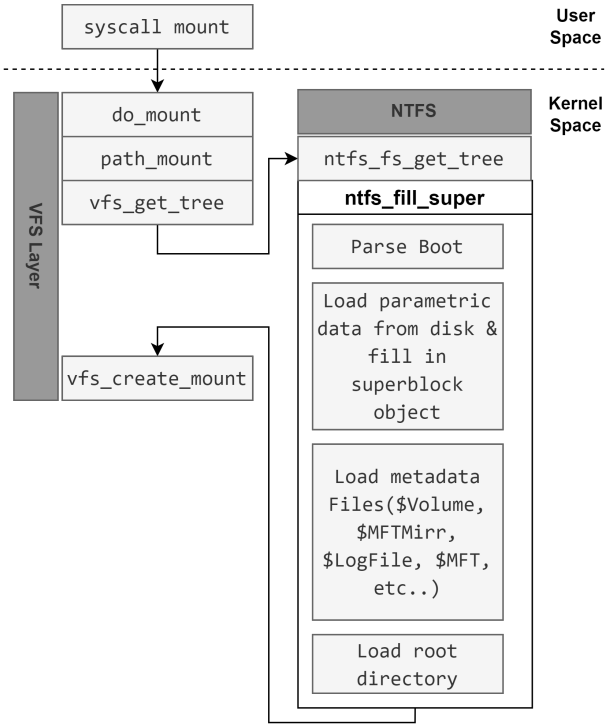

Fig. 3. The simplified trace of mounting an NTFS disk.

from the master file table. Finally, it reads the root directory of the NTFS file system from disk. All these loaded data will be filled into a superblock structure, i.e., `ntfs_sb_info`.

In this section, we conduct case studies against three representative Type I bugs, i.e., `0b6604`, `c1ca8e`, and `e19c62`. The root causes for these three bugs are different. However, the system will crash once the image is mounted. We will delve deeper in the following.

1) 0b66046: This bug results from a null pointer dereference due to an implementation error. Specifically, as we mentioned in Fig. IV-B, the first step of `ntfs_fill_super` is parsing the partition parse boot, which is implemented through a function named `ntfs_init_from_boot`, which is implemented in Listing 2.

```
static int ntfs_init_from_boot(struct super_block
    * sb, u32 sector_size, u64 dev_size) {
  // some operations
  sbi -> record_size = record_size = boot ->
      record_size < 0 ?
  1 << (-boot -> record_size) :
  (u32) boot -> record_size << sbi -> cluster_bits
      ;

  if (record_size > MAXIMUM_BYTES_PER_MFT)
    goto out;

  sbi -> record_bits = blksize_bits(record_size);
  // some operations
}

/* assumes size > 256 */
static inline unsigned int blksize_bits(unsigned
    int size) {
  unsigned int bits = 8;
  do {
    bits++;
    size >>= 1;
  } while (size > 256);
```

```
21    return bits;
22  }
```

Listing 2. The implementation of ntfs_init_from_boot

As we can see, the boot record size (boot->record_size) is read from disk at L3, which will then be passed to function blksize_bits through a variable named record_size. The comment at L14 says that record_size should be larger than 256. However, before passing record_size to blksize_bits, there is only a verification to identify if it is greater than a maximum limit (L7). In other words, if record_size is less than 256, function blksize_bits will only return 8 to record_size (L10), smaller than the ordinary situations. In the following stages during mounting the disk, a pointer will be shifted left record_bits bits and shifted right several bits. Because the variable record_bits will be only 8 when record_size is smaller than 256, the value of the pointer will be so small that it will point to a invalid address, leading to a null pointer dereference. According to the log file, the value of the pointer, i.e., the address, is only 0000000000000158, which is an invalid memory address.

Therefore, the patch will limit the range of record_size. In other words, its acceptable range should be limited by not only a maximum value (MAXIMUM_BYTES_PER_MFT at L7 in Listing 2), but also by a minimum value. Listing 3 shows the corresponding patch. As we can see, we set a minimum limit as SECTOR_SIZE, because a boot record in NTFS always includes the first sector of the disk image, whose size is SECTOR_SIZE, i.e., 512 bytes.

```
1  diff --git a/fs/ntfs3/super.c b/fs/ntfs3/super.c
2  index d72a27abf1c83..af9b7947df64e 100644
3  --- a/fs/ntfs3/super.c
4  +++ b/fs/ntfs3/super.c
5  @@ -814,7 +814,7 @@ static int ntfs_init_from_boot
       (struct super_block *sb, u32 sector_size,
6                              : (u32)boot->record_size
7                                    << sbi->
                                        cluster_bits;
8
9  -     if (record_size > MAXIMUM_BYTES_PER_MFT)
10 +     if (record_size > MAXIMUM_BYTES_PER_MFT ||
       record_size < SECTOR_SIZE)
11           goto out;
12
13       sbi->record_bits = blksize_bits(record_size);
```

Listing 3. Patch to the bug in §IV-B1

2) e19c627: Commit e19c627 is related to another Type I bug. It will eventually lead to an out-of-bound access due to a missing on integer overflow check.

Specifically, this bug lies in the function mi_enum_attr. It is an enumerator on file attributes of the disk image. As we mentioned in §II-A2, the MFT data structure centrally maintains attributes of files, whose detailed implementation is upon a struct, named ATTRIB, which is implemented in Listing 4.

```
1  struct ATTRIB {
2    enum ATTR_TYPE type;  // 0x00: The type of this
         attribute.
3    __le32 size;     // 0x04: The size of this
         attribute.
4    u8 non_res;    // 0x08: Is this attribute non-
         resident?
5    u8 name_len;     // 0x09: This attribute name
         length.
6    __le16 name_off;  // 0x0A: Offset to the
         attribute name.
7    __le16 flags;     // 0x0C: See ATTR_FLAG_XXX.
8    __le16 id;     // 0x0E: Unique id (per record).
9
10   union {
11     struct ATTR_RESIDENT res;      // 0x10
12     struct ATTR_NONRESIDENT nres; // 0x10
13   };
14 };
```

Listing 4. The data structure maintains attributes of files

The field named size (L3) records the size of this struct. Because this struct lies on the disk one by one adjacently, through reading the field size of the current struct, the system is able to get the address of the next struct. Intuitively, an illegal access tends to happen if the size is too big. In mi_enum_attr, the size of a struct will be directly assigned to a variable named asize. To get the next struct's offset on disk, asize is added with the offset of the current struct, dubbed off. If size is too big, accessing the next struct tends to fall out of the disk image. This case is considered and checked in L7 of Listing 5.

```
1  struct ATTRIB *mi_enum_attr(struct mft_inode *mi,
       struct ATTRIB *attr)
2  {
3     u32 t32, off, asize;
4     asize = le32_to_cpu(attr->size);
5
6     /* Check boundary. */
7     if (off + asize > used)
8         return NULL;
9     ...
10 }
```

Listing 5. Boundary check of asize

But if size is big enough, off + asize will overflow and generate a quite small number, leading to a failure on such a boundary check. Thus, there will be an out-of-bound read.

The patch for this bug is straightforward. It adds another check if an integer overflow happens on the addition as shown in Listing 6.

```
1  diff --git a/fs/ntfs3/record.c b/fs/ntfs3/record.c
2  index c8741cfa421fe..66eb11e0965ef 100644
3  --- a/fs/ntfs3/record.c
4  +++ b/fs/ntfs3/record.c
5  @@ -220,6 +220,11 @@ struct ATTRIB *mi_enum_attr(
       struct mft_inode *mi, struct ATTRIB *attr)
6                return NULL;
7             }
8
9  +        if (off + asize < off) {
10 +            /* overflow check */
11 +            return NULL;
12 +        }
13 +
14            attr = Add2Ptr(attr, asize);
15            off += asize;
16        }
```

Listing 6. Patch to the bug in §IV-B2

3) c1ca8ef: Differing from the above two cases that are logical bugs, this one is an unhandled corner case.

Specifically, this bug also happens in the function ntfs_fill_super, part of which is shown in Listing 7. As we can see, at L1, it invokes ntfs_iget5 to retrieve an inode, which will then be dispatched into d_make_root (an API of Linux kernel's VFS subsystem) to create the root directory of the mounting disk.

```
1       inode = ntfs_iget5(sb, &ref, &NAME_ROOT);
2       if (IS_ERR(inode)) {
3           ntfs_err(sb, "Failed to load root.");
4           err = PTR_ERR(inode);
5           goto out;
6       }
7
8       sb->s_root = d_make_root(inode);
```

Listing 7. A code snippet of `ntfs_fill_super`

The `inode` is a kernel struct that represents a file or a directory in Linux kernel. One of the fields of this struct is named `i_op`. It is a pointer pointing to a function table that is composed of file operation handlers like `rename`, `mkdir` and `unlink`. When initiating an `inode`, the `i_op` will be firstly initiated as a NULL pointer. Then, there is a verification that can be jumped over by constructing arguments. If it is, the control flow will be directed to a label where the `inode` will be returned directly, without assigning a concrete value for `i_op`. Thus, the returned `inode` will carry an empty `i_op` and be passed to `d_make_root` (L8 of Listing 7), within which some file operations will be performed by dereferencing `i_op`, the NULL pointer. An invalid memory access is exploited.

The bug fix takes the value of `i_op` into consideration, as illustrated in Listing 8.

```
1   diff --git a/fs/ntfs3/super.c b/fs/ntfs3/super.c
2   index ff70e2a5f3acb..1e2c04e48f98f 100644
3   --- a/fs/ntfs3/super.c
4   +++ b/fs/ntfs3/super.c
5   @@ -1286,9 +1286,9 @@ load_root:
6        ref.low = cpu_to_le32(MFT_REC_ROOT);
7        ref.seq = cpu_to_le16(MFT_REC_ROOT);
8        inode = ntfs_iget5(sb, &ref, &NAME_ROOT);
9   -    if (IS_ERR(inode)) {
10  +    if (IS_ERR(inode) || !inode->i_op) {
11           ntfs_err(sb, "Failed to load root.");
12  -        err = PTR_ERR(inode);
13  +        err = IS_ERR(inode) ? PTR_ERR(inode) : -
        EINVAL;
14           goto out;
15       }
```

Listing 8. Patch to the bug in §IV-B3

C. Case Study on Type II

As we mentioned in §IV-B, when mounting an NTFS image, Linux parses some structures and loads metadata which will then be filled into an NTFS superblock. However, the attributes of files will only be read during the corresponding file operations (like open or renaming). Therefore, even if a disk is mounted successfully, the system may also crash when some specific file operations are invoked. We categorize these bugs as Type II ones. Papora has successfully identified two Type II bugs, which are detailed in the following.

1) 54e4570: Commit `54e4570` is related to a Type II vulnerability[3], which will be triggered once updating attributes of a specific file, whose metadata is mutated by Papora. Fig. 4 shows the mutated attributes of the file. We can see that an attribute, named `NameLength`, has a value of 255.

The function `ni_create_attr_list` iterates file attributes in an NTFS image with `mi_enum_attr` in a for-loop, as shown at L9 of Listing 9. Then, it will copy the

[3]This one is not a bug because it can lead to an out-of-bound write.

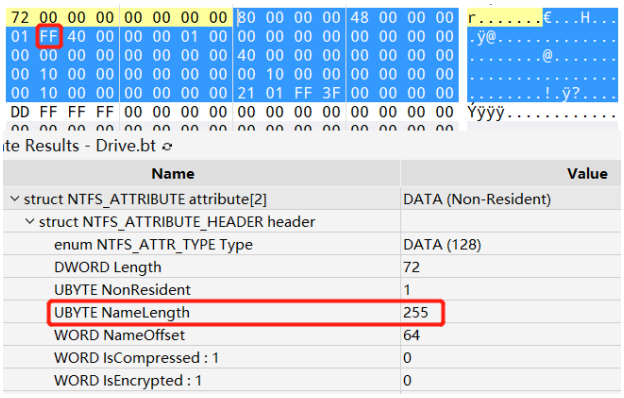

Fig. 4. The mutated file attributes of the vulnerability in §IV-C1.

attributes one by one to the heap memory. All attributes of a file are read by `mi_enum_attr`, but it fails to check the attribute `NameLength`. If `NameLength` is larger than the remaining allocated heap memory, a heap out-of-bound access will happen.

```
1   int ni_create_attr_list(struct ntfs_inode * ni) {
2       ...
3       le = kmalloc(al_aligned(rs), GFP_NOFS);
4       if (!le) {
5         err = -ENOMEM;
6         goto out;
7       }
8       ...
9       for (; (attr = mi_enum_attr( & ni -> mi, attr)
            ); le = Add2Ptr(le, sz)) {
10        sz = le_size(attr -> name_len);
11        le -> type = attr -> type;
12        le -> size = cpu_to_le16(sz);
13        le -> name_len = attr -> name_len;
14        le -> name_off = offsetof(struct
              ATTR_LIST_ENTRY, name);
15
16        if (attr -> name_len)
17          memcpy(le -> name, attr_name(attr), sizeof
                (short) * attr -> name_len);
18      }
19    }
20  }
```

Listing 9. The implementation of `ni_create_attr_list`

To trigger this vulnerability, the program shown in Listing 10 provides a feasible exploit. Specifically, the program invokes `setxattr` at L16, setting attributes of a file, which eventually invokes the `ni_create_attr_list` to exploit the vulnerability.

```
1   v9 = syscall(SYS_open, (long)v8, 2, 0);
2   syscall(SYS_read, (long)v9, (long)v0, 5195);
3   syscall(SYS_unlink, (long)v3);
4   syscall(SYS_truncate, (long)v6, 4367);
5   syscall(SYS_unlink, (long)v7);
6   syscall(SYS_symlink, (long)v2, (long)v10);
7   syscall(SYS_lstat, (long)v2, (long)v1);
8   syscall(SYS_setxattr, (long)v2, (long)v12, (long)
        v11, 127, 1);
9   syscall(SYS_pread64, (long)v9, (long)v0, 6806,
        299);
10  syscall(SYS_listxattr, (long)v10, (long)v1, 5210);
11  syscall(SYS_removexattr, (long)v4, (long)v13);
12  syscall(SYS_removexattr, (long)v2, (long)v14);
13  v15 = syscall(SYS_open, (long)v4, 2, 0);
14  syscall(SYS_listxattr, (long)v5, (long)v1, 5836);
15  syscall(SYS_utimes, (long)v5, (long)v1);
16  syscall(SYS_setxattr, (long)v2, (long)v17, (long)
        v16, 11, 1);
17  syscall(SYS_lstat, (long)v2, (long)v1);
```

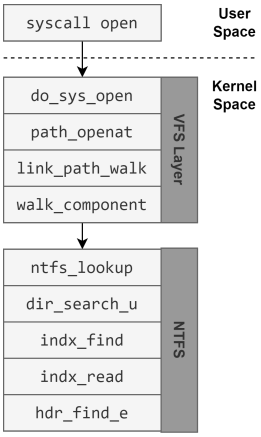

Fig. 5. The trace of the exploitation on the bug in §IV-C2.

```
18  syscall(SYS_pwrite64, (long)v9, (long)v1, 1772,
        434);
```

Listing 10. The program to exploit the vulnerability in §IV-C1

To fix this bug, as we mentioned above, we should pay attention to the `NameLength` field. The corresponding patch is shown in Listing 11.

```
1   diff --git a/fs/ntfs3/record.c b/fs/ntfs3/record.c
2   index 66eb11e0965ef..a952cd7aa7a4b 100644
3   --- a/fs/ntfs3/record.c
4   +++ b/fs/ntfs3/record.c
5   @@ -265,6 +265,11 @@ struct ATTRIB *mi_enum_attr(
        struct mft_inode *mi, struct ATTRIB *attr)
6                   if (t16 + t32 > asize)
7                       return NULL;
8
9   +       if (attr->name_len &&
10  +           le16_to_cpu(attr->name_off) + sizeof(short
        ) * attr->name_len > t16) {
11  +         return NULL;
12  +       }
13  +
14                  return attr;
15          }
```

Listing 11. The patch to the vulnerability in §IV-C1

2) 4d42ecd: This bug is an out-of-bound read that is triggered by an `open` system call. From the log file, we can conclude the trace as shown in Fig. 5. The bug is triggered in `hdr_find_e`.

There is a structure named index buffer in the NTFS disk image. It is composed of a header and several entries. The header is defined by a struct, named `INDEX_BUFFER`, as shown in Listing 12. The field `ihdr` holds some metadata of the buffer itself, like the length of the buffer and how many entries are used already.

```
1   struct INDEX_BUFFER {
2     struct NTFS_RECORD_HEADER rhdr; // 'INDX'
3     ...
4     struct INDEX_HDR ihdr; // stores metadata
5   };
```

Listing 12. A code snippet of `INDEX_BUFFER` struct

The function `hdr_find_e` conducts a binary search in the index buffer to find a certain entry, i.e., a specific file index. Note that the buffer is allocated by the kernel and its size is calculated from a variable read from disk, named as `index_block_size`. Specifically, the binary search adopts `ihdr->used` to calculate the end of the buffer. If it's inconsistent with the result buffer size calculated from `index_block_size`, e.g., larger than the allocated size of index buffer, the binary search will access the outside.

The aim of the corresponding patch is to make sure the search cannot access the outside. As we can see from Listing 13, the `bytes` at L10 is the index buffer allocation size. `offsetof(struct INDEX_BUFFER, ihdr) + ib->ihdr.used` calculates the index buffer size from `used`. To this end, it guarantees the access should always be limited within the allocated buffer.

```
1   diff --git a/fs/ntfs3/index.c b/fs/ntfs3/index.c
2   index 613036f9c6e66..bc656868cf8a8 100644
3   --- a/fs/ntfs3/index.c
4   +++ b/fs/ntfs3/index.c
5   @@ -1017,6 +1017,12 @@ ok:
6                   err = 0;
7           }
8
9   + /* check for index header length */
10  + if (offsetof(struct INDEX_BUFFER, ihdr) + ib->
        ihdr.used > bytes) {
11  +   err = -EINVAL;
12  +   goto out;
13  + }
14  +
15          in->index = ib;
16          *node = in;
```

Listing 13. Patch to the bug in §IV-C2

V. Lessons Learned

As we categorize the root causes of bugs identified by Papora in §IV-A2, there are a couple of things we learned from those findings, which we recommend file system developers to follow.

First of all, **user-controllable data should always be treated as untrusted input**. When a file system image is mounted, the Linux VFS routes the *mount* request to the corresponding file system handler which reads data from the disk and parses them in the memory. If the file system image contains fields which would be used to derive an array index or memory pointer, those fields could be easily crafted to trigger out-of-bounds access in kernel space, which leads to system crash or even local privilege escalation. Fortunately, the specifications of most file systems are well-documented. A file system developer could follow the specification to strictly check every single chunk of data read from the disk.

Secondly, **type confusion issues should be paid more attention**. In all Unix-like systems, an `inode` is used to describe a file, a directory, or other file system objects. However, the handling logic to processing a file could be totally different to process a directory. If an object is wrongly interpreted as another, unexpected behaviors occur. Actually, programming languages without memory safety (e.g., C and C++) are prone to weaknesses in this type. The Linux kernel is also evolving into an operating system with languages enforcing memory safety [24].

Last but not least, **conducting a high code coverage fuzzing testing for file systems is necessary.** To the best of our knowledge, no off-the-shelf fuzzer can efficiently fuzz

a new file system in the Linux kernel. Meanwhile, this also indicates that new file systems could be great targets for security researchers but not the best choices for users. In particular, the `get_tree()` handler of each file system (e.g., `ntfs_fs_get_tree()` of NTFS3) would be a good entry point for testing funny file system images. As illustrated in Fig. 3, the functions to parse data retrieved from disk (e.g., `ntfs_init_from_boot()`) may miss some important validation logic. Moreover, each file system has a handler (e.g., `ntfs_lookup()` of NTFS3) to search a file while handling an `open` system call (see Fig. 5). Security researchers could craft data related to the `lookup` procedure in a mutated disk image and see if that would cause out-of-bounds access.

VI. Related Work

Fuzzing has been proven effective in finding vulnerabilities in various softwares and kernel binaries, including file systems. Vulnerabilities in file systems can be exploited due to two orthogonal root causes, which are often taken as targets for fuzzers, i.e., *disk images* and *file operation-specific system calls*. Most existing fuzzers against file systems target either the former one [4]–[6], or the later one [17], [18], [25], [26]. For example, kAFL implemented by S. Schumilo et al. [6] has improved the AFL [4] to specifically target kernels. They have evaluated the kAFL across multiple operating systems by only mutating the disk images. Contrarily, the KRACE focuses on multi-thread vulnerabilities, which must be triggered by a certain sequence of file operations. Some work [19], [20], [27], [28], including Papora, takes both factors into consideration. However, Papora targets another critical but close-source file system, NTFS, and identifies a dozen of vulnerabilities that are acknowledged by the Linux kernel.

VII. Discussion

Unique Challenges in Fuzzing NTFS File Systems. Compared to fuzzing other file systems, fuzzing NTFS images has some unique challenges, which can be concluded mainly in twofold. On the one hand, non-transparency in both terms of implementation and documentation hinders implementing a fuzzer. There is no official implementation released, and its so-called official documentation lacks lots of technical details. Therefore, we have to manually compare multiple third-party releases and their corresponding documentations, and cross-reference which implementation is adopted by Microsoft. On the other hand, such a non-transparency still occurs in checksum validation. For example, the OEM field of the BPS should start with "NTFS " (NTFS + 4 spaces). However, NTFS still requires the bytes per sector must be greater than 512 bytes and be a power of 2. Such constraints are not well documented in any documentation and we have to manually dig them up from source code of third-party releases.

Necessity of Implementing the Parser. There were several NTFS parsers, e.g., NTFS-3G and Linux legacy NTFS. However, they are not sufficient for supporting a fuzzer like Papora. Specifically, traditional parsers like NTFS-3G is responsible for validating the integrity of the given NTFS image. If something goes wrong, like a piece of problematic checksum, the parser will not parse and load the image at all. Fuzzing, however, will constantly mutate the metadata of the image to try to figure out new bugs. Such deliberately introducing corrupted metadata will invalidate traditional parsers. Therefore, the parser in Papora can not only parse the given image, but also automatically recover corrupted metadata, like recalculating checksums. Moreover, it is lighter than traditional ones which are typically maintained for several years. Therefore, the parser in Papora is not the reinvented wheel.

Advantages over Other Fuzzers. Papora has more advantages than other potential fuzzers against file systems. For example, fuzzers specifically designed for file systems, like Janus and Hydra, cannot parse an NTFS image and conduct the following analysis. Syzkaller could be usable in testing an NTFS image, because it mutates the program consisting of file operations to examine if any vulnerabilities can be triggered. However, under the same environment as Papora, we ran Syzkaller, with advanced options (like KCOV and SyzLang) enabled, for 3 weeks and no valid results could be obtained. We speculate that this phenomenon can be explained by the experimental results shown in Table I. For bugs we have identified, 10 out of 12 are due to buggy images. Only 2 of them can be triggered by a certain program, while it still requires a mutated disk as prerequisites. Therefore, only composing a series of random file operations cannot effectively identify bugs embedded in NTFS images, which forces us to find alternative methods and develop Papora.

VIII. Conclusion

In summary, we have proposed a fuzzer, named Papora, specifically targeting NTFS images. We have released the two core components of Papora, i.e., the NTFS parser, and the LKL that has been ported to the latest Linux kernel with KASAN integrated. Based on the efficiency and effectiveness of Papora, we have identified 3 assigned CVE 0-day vulnerabilities and 9 severe bugs within the latest release of the Linux kernel. All of them are confirmed by Linux maintainers and the corresponding patches of 9 out of them are merged into upstreams. For these identified bugs and vulnerabilities, we have conducted a thorough empirical study including case studies on representative cases. Finally, based on our investigations on those loopholes and exploits, we summarized a set of best practices for developers and security researchers.

Acknowledgment

We specially thank all anonymous reviewers and our shepherd for their valuable suggestions that significantly improve the quality of this paper.

REFERENCES

[1] New Technology File System (NTFS). [Online]. Available: https://en.wikipedia.org/wiki/NTFS

[2] Jan 2023. [Online]. Available: https://github.com/sslab-gatech/janus

[3] Jan 2023. [Online]. Available: https://github.com/sslab-gatech/hydra

[4] M. Zalewski. (2023) AFL project. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[5] Ribose. (2023) FuzzBSD project. [Online]. Available: https://github.com/riboseinc/fuzzbsd

[6] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "{kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 167–182.

[7] The Kernel Address Sanitizer (KASAN). [Online]. Available: https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html

[8] LKL: Linux Kernel Library. [Online]. Available: https://lkl.github.io/

[9] The Linux NTFS filesystem driver. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/ntfs.rst

[10] Captive: The first free NTFS read/write filesystem for GNU/Linux. [Online]. Available: http://www.jankratochvil.net/project/captive/

[11] ntfs-3g. [Online]. Available: https://github.com/tuxera/ntfs-3g

[12] ntfs3. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/fs/ntfs3?

[13] Paragon Software. [Online]. Available: https://www.paragon-software.com/

[14] Encrypting File System. [Online]. Available: https://en.wikipedia.org/wiki/Encrypting_File_System#Accessing_private_key_via_password_reset

[15] Organization of an NTFS Volume. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc781134(v=ws.10)#organization-of-an-ntfs-volume

[16] libFuzzer – a library for coverage-guided fuzz testing. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[17] D. Jones. (2023) Trinity project. [Online]. Available: https://github.com/kernelslacker/trinity

[18] Google. (2023) syzkaller project. [Online]. Available: https://github.com/google/syzkaller

[19] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 818–834.

[20] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding bugs in file systems with an extensible fuzzing framework," *ACM Transactions on Storage (TOS)*, vol. 16, no. 2, pp. 1–35, 2020.

[21] CWE-843:Access of Resource Using Incompatible Type ('Type Confusion'). [Online]. Available: https://cwe.mitre.org/data/definitions/843.html

[22] E. Lo. fs/ntfs3: Validate MFT flags before replaying logs. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/commit/?id=467333af2f7b95eeaa61a5b5369a80063cd971fd

[23] CWE-670:Always-Incorrect Control Flow Implementation. [Online]. Available: https://cwe.mitre.org/data/definitions/670.html

[24] Linus Torvalds: Rust will go into Linux 6.1. [Online]. Available: https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/

[25] NCC Group. (2023) TriforceAFL project. [Online]. Available: https://github.com/nccgroup/TriforceAFL

[26] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1643–1660.

[27] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 147–161.

[28] T. Groß, T. Schleier, and T. Müller, "Refuzz-structure aware fuzzing of the resilient file system (refs)," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 589–601.